

Multiparty Session Actors

Rumyana Neykova Nobuko Yoshida

Imperial College London

Actor coordination armoured with a suitable protocol description language has been a pressing problem in the actors community. We study the applicability of multiparty session type (MPST) protocols for verification of actor programs. We incorporate sessions to actors by introducing minimum additions to the model such as the notion of actor roles and protocol mailbox. The framework uses Scribble, which is a protocol description language based on multiparty session types. Our programming model supports actor-like syntax and runtime verification mechanism guaranteeing type-safety and progress of the communicating entities. An actor can implement multiple roles in a similar way as an object can implement multiple interfaces. Multiple roles allow for inter-concurrency in a single actor still preserving its progress property. We demonstrate our framework by designing and implementing a session actor library in Python and its runtime verification mechanism.

1 Introduction

The actor model is (re)gaining attention in the research community and in the mainstream programming languages as a promising concurrency paradigm. Unfortunately, in spite of the importance of message passing mechanisms in the actor paradigm, the programming model itself does not ensure correct sequencing of interactions between different computational processes. This is a serious problem when designing safe concurrent and distributed systems in languages with actors.

To overcome this problem, we need to solve several shortcomings existing in the actor programming models. First, although actors often have multiple states and complex policies for changing states, no general-purpose specification language is currently in use for describing actor protocols. Second, a clear guidance on actor discovery and coordination of distributed actors is missing. As a study published in [13] reveals, this leads to adhoc implementations and mixing the model with other paradigms which weaken its benefits. Third, no verification mechanism (neither static nor dynamic) is proposed to ensure correct sequencing of actor interactions. Most actor implementations provide static typing within a single actor, but the communication between actors – the complex communication patterns that are most likely to deadlock – are not checked.

We tackle the aforementioned challenges by studying applicability of multiparty session types (MPST) verification and its practical incarnation, the protocol description language Scribble, to actor systems. Recent works from [8, 6] prove suitability of MPST for dynamic verification of real world complex protocols [9]. The verification mechanism is applied to large cyberinfrastructure, but checks are restricted only to MPST primitives. In this paper, we take MPST verification one step further and apply it to an actor model by extending it with creation and management of communication contexts (protocols).

Our programming model is grounded on three design ideas: (1) use Scribble protocols and their relation to finite state machines for specification and runtime verification of actor interactions; (2) augment actor messages and their mailboxes dynamically with protocol (role) information; and (3) propose an algorithm based on virtual routers (protocol mailboxes) for the dynamic discovery of actor mailboxes within a protocol. We implement a session actor library in Python to demonstrate the applicability of the

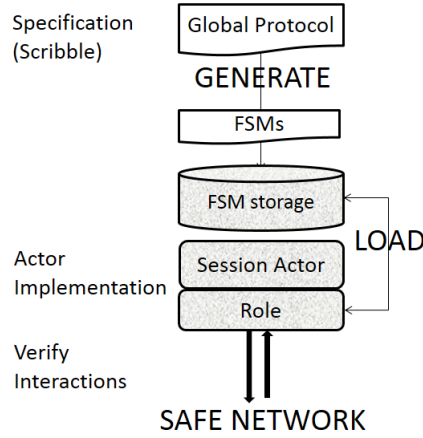


Figure 1: Development methodology

approach. To the best of our knowledge, this is the first design and implementation of session types and their dynamic verification toolchain in an actor library.

2 Multiparty Session Actor Programming

2.1 Overview of Multiparty Session Actor Framework

The standard development methodology for MPST verification is illustrated in Fig. 1, and the contributions of this work are darked. Distributed protocols are specified in Scribble, which collectively defines the admissible communication behaviours. Scribble tool can algorithmically project any global conversation to the specifications of its endpoints, generating finite state machines (FSMs). This is different from previous works [8, 6], where Scribble compiler produces local Scribble protocols and FSMs are generated on the fly at runtime. This difference is important for the applicability of our framework to general actor programming.

Distributed components are realised as session actors associated with one or more of the generated FSMs. The association is done through annotating the actor type (class) and receive messages (methods) with protocol information. Annotating the actor type with protocol information results in registering the type for a particular role. When a session is started, a join message is sent (in a round-robin style) to all registered actors. When a join message is received, the generated FSM is loaded into the actor role and all subsequent messages on that protocol (role) are tracked and checked. Message receive is delegated to the appropriate FSM via pattern matching on the protocol id, contained in the message. If all actors messages comply to their assigned FSMs, the whole communication is guaranteed to be safe. If participants do not comply, violations (such as deadlocks, communication mismatch or breakage of a protocol) are detected and delegated to a Policy actor.

2.2 Warehouse Management Protocol in Session Actors

To illustrate and motivate central design decisions of our model, we present the buyer-seller protocol from [5] and extend it to a full warehouse management scenario. A warehouse consists of multiple customers communicating to a warehouse provider. It can be involved in purchase protocol (with customers), but can also be involved in a loaded protocol with dealers to update its storage.

<pre> 1 global protocol Purchase 2 (role B, role S, role A) 3 { 4 login(string:user) from B to S; 5 login(string:user) from S to A; 6 authenticate(string:token) from A to B, S; 7 choice at B 8 {request(string:product) from B to S; 9 (int:quote) from S to B;} 10 or 11 {buy(string:product) from B to S 12 delivery(string) from S to B; } 13 or 14 {quit() from B to S; }} 15 global protocol StoreLoad 16 (role D, role S) 17 { 18 rec Rec{ 19 choice at S 20 {request(string:product, int:n) from S to D;) 21 put(string:product, int:n) from D to S; 22 continue Rec;} 23 or 24 {quit() from S to D; 25 acc() from D to S;}}}</pre>	<pre> @protocol(c, Purchase, seller, buyer, auth) @protocol(c1, StoreLoad, seller, dealer) class Warehouse(SessionActor): @role(c, buyer) def login(self, user): c.auth.send.login(user) @role(c, buyer) def buy(self, product): self.purchaseDB[product]-=1; c.seller.send.delivery(product.details) self.become(update, product) @role(c, buyer) def quit(self): c.send.buyer.acc() @role(c1, self) def update(self, product): c1.dealer.send.request(product, n) @role(c1, dealer) def put(self, c, product): self.purchaseDB[product]+=1:</pre>
---	--

Figure 2: Global protocols in Scribble

Figure 3: Session Actor for warehouse

Scribble Protocol. Fig. 2 shows the interactions between the entities in the system written as a Scribble protocol. There are purchase and storeload protocols, involving three (a Buyer (B), a Seller (S) and an Authenticator (A)) and two (a Store (S), a Dealer (D)) parties, respectively. At the start of a purchase session, B sends login details to S, which delegates the request to an Authentication server. After the authentication is completed, B sends a request quote for a product to S and S replies with the product price. Then B has a choice to ask for another product, to proceed with buying the product, or to quit. By buying a product the warehouse decreases the product amount it has in the store. When a product is out of stock, the warehouse sends a product request to a dealer to load the store with n numbers of that product (following the storeload protocol). The reader can refer to [11] for the full specification of Scribble syntax.

Challenges. There are several challenging points to implement the above scenario. First, a warehouse implementation should be involved in both protocols, therefore it can play more than one role. Second, initially the user does not know the exact warehouse it is buying from, therefore the customer learns dynamically the address of the warehouse. Third, there can be a specific restriction on the protocol that cannot be expressed as system constants (such as specific timeout depending on the customer). The next section explains implementations of Session Actors in more details.

Session Actor. Fig. 3 presents an implementation of a warehouse service as a single session actor that keeps the inventory as a state (*self.purchaseDB*). Lines 1- 2 annotate the session actor class with two protocol decorators – *c* and *c1* (for seller and store roles respectively). *c* and *c1* are accessible within the warehouse actor and are holders for mailboxes of the other actors, involved in the two protocols. All message handlers are annotated with a role and for convenience are implemented as methods. For

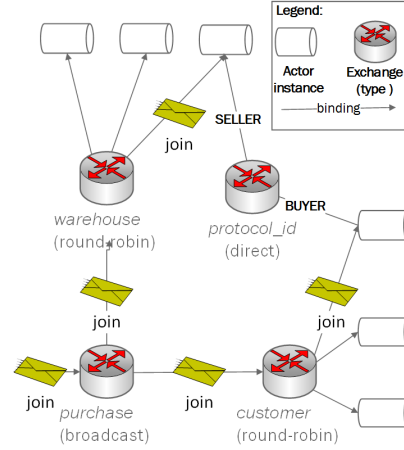


Figure 4: Organising Actors into protocols

example, the login method (Line 6) is invoked when a *login* message (Line 4, Fig. 2) is sent. The role annotation for *c* (Line 5) specifies the sender to be *buyer*. The handler body continues following Line 5, Fig. 2 - sending a *login* message via the *send* primitive to the session actor, registered as a role *auth* in the protocol of *c*. Value *c.auth* is initialised with the *auth* actor mailbox as a result of the actor discovery mechanism (explained in the next section). The handling of *authenticate* (Line 6, Fig. 2) and *request* (Line 8, Fig. 2) messages is similar, so we omit it and focus on the *buy* handler (Line 10- 13), where after sending the delivery details (Line 12), the warehouse actor sends a message to itself (Line 13) using the primitive *become* with value *update*. Value *update* is annotated with another role *c1*, but has as a sender *self*. This is the mechanism used for switching between roles within an actor. Update method (Line 20- 21, Fig. 3) implements the request branch (Line 20-22, Fig. 2) of the *StoreLoad* protocol - sending a request to the *dealer* and handling the reply via method *put*. The correct order of messages is verified by the FSM attached to *c* and *c1*. As a result, errors such as calling *put* before *update* or executing two consecutive updates, will be detected as invalid.

3 Implementation of Multiparty Session Actors

AMQP in a Nutshell Distributed actor library. We have implemented the multiparty session actors on top of Celery [3] (distributed message queue in Python) with support for distributed actors. Celery uses advanced message queue protocol (AMQP 0-9-1 [2]) as a transport. The reason for choosing AMQP network as base for our framework is that AMQP middleware shares a similar abstraction with the actor programming model, which makes the implementation of distributed actors more natural. AMQP model can be summarised as follow: messages are published by producers to entities, called exchanges (or mailboxes). Exchanges then distribute message copies to queues using rules called bindings. Then AMQP brokers (virtual routers) deliver messages to consumers subscribed to queues. Distributed actors are naturally represented in this context using the abstractions of exchanges. Each actor type is represented in the network as an exchange and is realised as a consumer subscribed to a queue based on a pattern matching on the actor id. Message handlers are implemented as methods on the actor class.

Our distributed actor discovery mechanism draws on the AMPQ abstractions of exchanges, queues and binding, and our extensions to the actor programming model are built using Python advanced abstraction capabilities: two main capabilities are coroutines (for realising the actors inter-concurrency) and decorators (for annotating actor types and methods).

Actor roles. A key idea is each role to run in a virtual thread of an actor (using Python coroutines/green threads). We annotate methods, implementing part of a protocol, with a role decorator. Roles are scheduled cooperatively. This means that at most one role can be active in a session actor at a time. A role is activated when a message is received and ready to be processed. Switching between roles is done via the *become* primitive (as demonstrated in Fig. 3), which is realised as sending a message to the internal queue of the actor.

Actors discovery. Fig. 4 presents the network setting (in terms of AMQP objects) for realising the actor discovery for *buyer* and *seller* of the protocol *Purchase*. For simplicity, we create the actor exchanges on starting of the actor system – round-robin exchange per actor type (*warehouse* and *customer* in Fig. 4) and broadcast exchange per protocol type (*purchase* in Fig. 4). All spawned actors alternate to receive messages addressed to their type exchange. Session actors are registered for roles via the protocol decorator and as a result their type exchange is bound to the protocol exchange (Line 1 in Fig. 3 binds *warehouse* to *purchase* in Fig. 4).

We explain the workflow for actor discovery. When a protocol is started, a fresh protocol id and an exchange with that id are created. The type of the exchange is *direct*¹ (*protocol id* in Fig. 4). Then *join* message is sent to the protocol exchange and delivered to one actor per registered role (*join* is broadcasted to *warehouse* and *customer* in Fig. 4). On *join*, an actor binds itself to the *protocol id* exchange with subscription key equal to its role (bindings *seller* and *buyer* in Fig. 4). When an actor sends a message to another actor within the same session (for example *c.buyer.send* in Fig. 3), the message is sent to the protocol id exchange (stored in *c*) and from there delivered to the *buyer* actor.

Order preservation through FSM checking. Whenever a message is received the actor internal loop dispatches the message to the role the message is annotated with. The FSM, generated from the Scribble compiler (as shown in Fig. 1), is loaded when an actor joins a session and messages are passed to the FSMs for checking before being dispatched to their handler. The FSM perform checks message labels (already part of the actor payload) and sender and receiver roles (part of the message binding key due to our extension). An outline of the monitor implementation can be also found in [6]. The monitor mechanism is an incarnation of the session monitor in [6] within actors.

4 Related and Future Work

There are several theoretical works that have studied the behavioural types for verifying actors [7, 4]. The work [4] proposes a behavioural typing system for an actor calculus where a type describes a sequence of inputs and outputs performed by the actor body, while [7] studies session types for a core of Erlang. On the practical side, the work [10] proposes a framework of three layers for actor programming - actors, roles and coordinators, which resembles roles and protocol mailboxes in our setting. Their specifications focus on QoS requirements, while our aim is to describe and ensure correct patterns of interactions (message passing). Scala actor library [1] (studied in [13]) supports FSM verification mechanism (through inheritance) and typed channels. Their channel typing is simple so that it cannot capture structures of communications such as sequencing, branching or recursions. These structures ensured by session types are the key element for guaranteeing deadlock freedom between multiple actors. In addition, in [1], channels and FSMs are unrelated and cannot be intermixed; on the other hand, in our approach, we rely on external specifications based on the choreography (MPST) and the FSMs usage is internalised (i.e. FSMs are automatically generated from a global type), therefore it does not affect program structures. To our best knowledge, no other work is linking FSMs, actors and choreographies in a single framework.

¹A direct type means that messages with routing key R are delivered to actors linked to the exchange with binding R.

As a future work, we plan to develop (1) extensions to other actor libraries to test the generality of our framework and (2) extensive evaluations of the performance overhead of session actors. The initial micro benchmark shows the overhead of the three main additions to our framework (FSM checking, actor type and method annotation) is negligible per message, see [12] (for example, the overhead of FSM checking is less than 2%). As actor languages and frameworks are getting more attractive, we believe that our framework would become useful for coordinating large-scale, distributed actor programs.

References

- [1] *Akka - scala actor library*. <http://akka.io/>.
- [2] *Advanced Message Queuing Protocol homepage*. <http://www.amqp.org/>.
- [3] *Celery*. <http://http://www.celeryproject.org/>.
- [4] Silvia Crafa: *Behavioural Types for Actor Systems*. arXiv:1206.1687.
- [5] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL'08*, ACM, pp. 273–284. Available at <http://dx.doi.org/10.1145/1328438.1328472>.
- [6] Raymond Hu, Rumyana Neykova, Nobuko Yoshida & Romain Demangeon (2013): *Practical Interruptible Conversations: Distributed Dynamic Verification with Session Types and Python*. In: *RV'13, LNCS 8174*, Springer, pp. 130–148. Available at http://dx.doi.org/10.1007/978-3-642-40787-1_8.
- [7] Dimitris Mostrous & Vasco Thudichum Vasconcelos (2011): *Session Typing for a Featherweight Erlang*. In: *COORDINATION, LNCS 6721*, Springer, pp. 95–109. Available at http://dx.doi.org/10.1007/978-3-642-21464-6_7.
- [8] Rumyana Neykova, Nobuko Yoshida & Raymond Hu (2013): *SPY: Local Verification of Global Protocols*. In: *RV'13, LNCS 8174*, Springer, pp. 358–363. Available at http://dx.doi.org/10.1007/978-3-642-40787-1_25.
- [9] *Ocean Observatories Initiative*. <http://www.oceanobservatories.org/>.
- [10] Shangping Ren, Yue Yu, Nianen Chen, Kevin Marth, Pierre-Etienne Poirot & Limin Shen (2006): *Actors, Roles and Coordinators - A Coordination Model for Open Distributed and Embedded Systems*. In: *COORDINATION, LNCS 4038*, Springer, pp. 247–265. Available at http://dx.doi.org/10.1007/11767954_16.
- [11] *Scribble project home page*. <http://www.scribble.org>.
- [12] *Online Appendix for this paper*. <http://www.doc.ic.ac.uk/~rn710/sactor>.
- [13] Samira Tasharofi, Peter Dinges & Ralph E. Johnson (2013): *Why Do Scala Developers Mix the Actor Model with other Concurrency Models?* In: *ECOOP, 7920*, Springer, pp. 302–326. Available at http://dx.doi.org/10.1007/978-3-642-39038-8_13.